**/**

# Assignment 1

1. Perform encryption, decryption using the following substitution techniques:
   1. Ceaser cipher Ans:

The Caesar Cipher is a simple encryption technique where each letter in a message is shifted by a fixed number of positions in the alphabet. For example, with a shift of 3, "A" becomes "D," "B" becomes "E," and so on. It's one of the oldest known ciphers and is easy to implement but also easy to break.

Code C++ : #include

<iostream> using namespace std;

string caesarCipherEncrypt(string text, int s) { string result = "";

for (int i = 0; i < text.length(); i++) { if (isupper(text[i]))

result += char(int(text[i] + s - 65) % 26 + 65);

else

}

result += char(int(text[i] + s - 97) % 26 + 97);

return result;

}

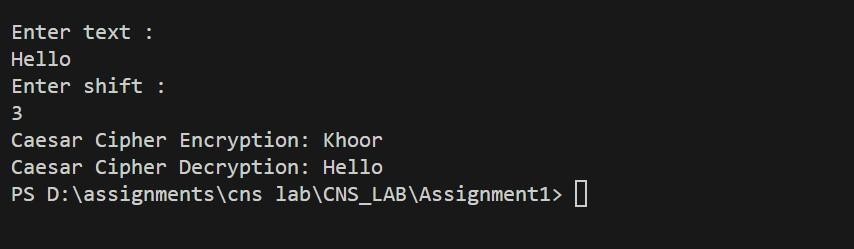
string caesarCipherDecrypt(string text, int s) { return caesarCipherEncrypt(text, 26 - s);

}

int main() { string text ; int shift; cin>>text; cin>>shift;

string encrypted = caesarCipherEncrypt(text, shift); string decrypted = caesarCipherDecrypt(encrypted, shift); cout << "Caesar Cipher Encryption: " << encrypted << endl; cout << "Caesar Cipher Decryption: " << decrypted << endl; return 0;

}

Output :

Advantages:

* Simplicity: Easy to understand and implement.
* Efficiency: Fast encryption and decryption. Disadvantages:
* Weak Security: Vulnerable to frequency analysis and brute-force attacks (only 25 possible shifts).
* Predictability: Does not change much between different texts.
  1. Playfair cipher Ans:

The Playfair Cipher is a digraph substitution cipher that encrypts pairs of letters. It uses a 5x5 matrix of letters created from a keyword. To encrypt, locate each letter pair in the matrix and swap or substitute based on their positions. It’s more secure than simple substitution ciphers because it encodes pairs of letters rather than individual letters.

C++ Code:

#include <iostream> #include <string> #include <vector> using namespace std;

void displayKeyMatrix(char keyMatrix[5][5]) { cout << "Playfair Cipher Key Matrix: \n"; for (int i = 0; i < 5; i++) { for (int j = 0; j

< 5; j++) {

cout << keyMatrix[i][j] << " ";

}

cout << endl;

}

cout << endl;

}

void generateKeyMatrix(string key, char keyMatrix[5][5]) { vector<bool> used(26, false); int x = 0, y = 0;

for (char &c : key) { if (c

== 'J') c = 'I'; if (!used[c -

'A']) { keyMatrix[x][y++]

= c; used[c - 'A'] = true;

if (y == 5) { y = 0; x++; }

}

}

for (char c = 'A'; c <= 'Z'; c++) { if (c == 'J') continue; if (!used[c - 'A']) {

keyMatrix[x][y++] = c; used[c - 'A'] = true;

if (y == 5) { y = 0; x++; }

}

}

}

pair<int, int> findPosition(char keyMatrix[5][5], char c) { if (c == 'J') c = 'I'; for (int i = 0; i < 5; i++) { for (int j = 0; j < 5; j++) { if (keyMatrix[i][j] == c) return {i, j};

}

}

return { -1, -1 };

}

string preprocessText(string text) { string processedText = ""; for (int i

= 0; i < text.length(); i++) { char first = text[i];

char second = (i + 1 < text.length()) ? text[i + 1] : 'X';

processedText += first;

if (first == second) { processedText += 'X';

} else if (i + 1 < text.length()) { processedText += second;

i++;

}

}

if (processedText.length() % 2 != 0) { processedText += 'X';

}

return processedText;

}

void displayPairs(string text) { cout << "Plaintext pairs:\n"; for (int i = 0; i < text.length(); i += 2) { cout << text[i] << text[i + 1] << " ";

}

cout << endl << endl;

}

string playfairCipher(string text, string key, bool encrypt = true) { char keyMatrix[5][5]; generateKeyMatrix(key, keyMatrix);

text = preprocessText(text);

if (encrypt) { displayKeyMatrix(keyMatrix);

displayPairs(text);

}

string result = "";

for (int i = 0; i < text.length(); i += 2) { char first = text[i]; char second = text[i + 1];

pair<int, int> pos1 = findPosition(keyMatrix, first); pair<int, int> pos2 = findPosition(keyMatrix, second);

if (pos1.first == pos2.first) {

result += keyMatrix[pos1.first][(pos1.second + (encrypt ? 1 : 4)) % 5]; result += keyMatrix[pos2.first][(pos2.second + (encrypt ? 1 : 4)) % 5];

} else if (pos1.second == pos2.second) {

result += keyMatrix[(pos1.first + (encrypt ? 1 : 4)) % 5][pos1.second]; result += keyMatrix[(pos2.first + (encrypt ? 1 : 4)) % 5][pos2.second];

} else {

result += keyMatrix[pos1.first][pos2.second]; result += keyMatrix[pos2.first][pos1.second]; }

}

return result;

}

string cleanDecryptedText(string decryptedText, string originalText) { string cleanText = ""; int j = 0;

for (int i = 0; i < decryptedText.length(); i++) {

if (decryptedText[i] == 'Z' && j < originalText.length() - 1 && originalText[j] == originalText[j + 1]) { j++;

} else {

cleanText += decryptedText[i];

j++;

}

}

if (cleanText.back() == 'X' && originalText.back() != 'X') { cleanText.pop\_back();

}

return originalText;

}

int main() { string text, key; cout << "Enter string: "; getline(cin,

text); cout << "Enter key: "; getline(cin, key);

for (auto &c : text) c = toupper(c); for (auto &c : key) c = toupper(c);

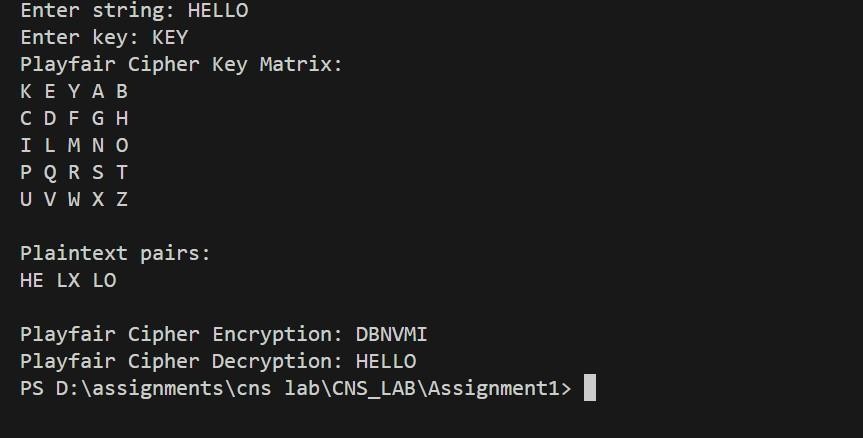
string encrypted = playfairCipher(text, key, true); cout << "Playfair Cipher Encryption: " << encrypted << endl;

string decrypted = playfairCipher(encrypted, key, false); decrypted = cleanDecryptedText(decrypted, text); cout << "Playfair Cipher Decryption: " << decrypted << endl;

return 0;

}

Output:



Advantages:

* Improved Security: More secure than Caesar Cipher as it encrypts digraphs (pairs of letters).
* Simplicity: Slightly more complex but still relatively easy to implement. Disadvantages:
* Key Management: Requires a good keyword and matrix setup.
* Vulnerability: Can still be broken with modern techniques like

frequency analysis of digraphs.

* 1. Hill Cipher

The Hill Cipher is a polygraphic substitution cipher that uses linear algebra. It encrypts blocks of text (usually 2x2 or 3x3 matrices) by multiplying them with a key matrix. The key matrix must be invertible for decryption. This method allows for more complex encryption compared to simple substitution ciphers.

Ans:

Code c++:

#include <iostream> #include <vector> #include <string>

#include <algorithm> // for gcd

using namespace std;

// Function to convert character to integer (A=0, B=1, ..., Z=25) int charToNum(char c) { return c - 'A';

}

// Function to convert integer to character (0=A, 1=B, ..., 25=Z) char numToChar(int n) { return 'A' + (n % 26);

}

// Function to multiply a matrix with a vector

vector<int> multiplyMatrix(vector<vector<int>> &matrix, vector<int> &vec, int size) {

vector<int> result(size); for (int i = 0; i < size; i++) { result[i] = 0;

for (int j = 0; j < size; j++) { result[i] += matrix[i][j] \* vec[j];

}

result[i] %= 26;

}

return result;

}

// Function to find the inverse of the key matrix (only works for 2x2 for simplicity)

vector<vector<int>> findInverseMatrix(vector<vector<int>> &matrix) { int det = matrix[0][0] \* matrix[1][1] - matrix[0][1] \* matrix[1][0]; det

= (det % 26 + 26) % 26; // Make sure determinant is positive and mod 26

// Check if determinant has an inverse (GCD must be 1) if ( gcd(det, 26) != 1) {

cout << "Matrix is not invertible!" << endl; return {};

}

// Multiplicative inverse of determinant mod 26 int invDet = -1; for (int i = 0; i < 26; i++) {

if ((det \* i) % 26 == 1) { invDet = i;

break;

}

}

// Create inverse matrix (for 2x2) vector<vector<int>> invMatrix(2, vector<int>(2));

invMatrix[0][0] = matrix[1][1] \* invDet % 26; invMatrix[1][1]

= matrix[0][0] \* invDet % 26; invMatrix[0][1] = - matrix[0][1] \* invDet % 26; invMatrix[1][0] = -matrix[1][0] \* invDet % 26;

// Adjust for negative values mod 26 for (int i = 0; i < 2; i++) { for (int j

= 0; j < 2; j++) {

invMatrix[i][j] = (invMatrix[i][j] + 26) % 26;

}

}

return invMatrix;

}

// Function to encrypt the plaintext

string encrypt(string plaintext, vector<vector<int>> &keyMatrix, int size) { string ciphertext = "";

// Add padding if needed to make the length a multiple of the matrix size if (plaintext.size() % size != 0) { plaintext += 'X';

}

for (int i = 0; i < plaintext.size(); i += size) { vector<int> messageVec(size); for (int j

= 0; j < size; j++) {

messageVec[j] = charToNum(plaintext[i + j]);

}

vector<int> encryptedVec = multiplyMatrix(keyMatrix, messageVec, size);

for (int j = 0; j < size; j++) {

ciphertext += numToChar(encryptedVec[j]);

}

}

return ciphertext;

}

// Function to decrypt the ciphertext

string decrypt(string ciphertext, vector<vector<int>> &invKeyMatrix, int size) {

string plaintext = "";

for (int i = 0; i < ciphertext.size(); i += size) { vector<int> encryptedVec(size);

for (int j = 0; j < size; j++) {

encryptedVec[j] = charToNum(ciphertext[i + j]);

}

vector<int> decryptedVec = multiplyMatrix(invKeyMatrix, encryptedVec, size); for (int j = 0; j < size; j++) {

plaintext += numToChar(decryptedVec[j]);

}

}

// Remove padding if present while (plaintext.back() == 'X') { plaintext.pop\_back();

}

return plaintext;

}

int main() { string plaintext;

int matrixSize = 2; // For simplicity, we'll use a 2x2 matrix vector<vector<int>> keyMatrix(matrixSize, vector<int>(matrixSize));

// Input the plaintext cout << "Enter the plaintext (only uppercase letters): "; cin >> plaintext;

// Input the key matrix

cout << "Enter the 2x2 key matrix (numbers between 0-25):" << endl; for (int i = 0; i < matrixSize; i++) { for (int j = 0; j < matrixSize; j++) { cin >> keyMatrix[i][j];

}

}

// Find the inverse of the key matrix

vector<vector<int>> invKeyMatrix = findInverseMatrix(keyMatrix); if (invKeyMatrix.empty()) {

return 1; // Exit if the matrix is not invertible

}

// Encrypt the plaintext

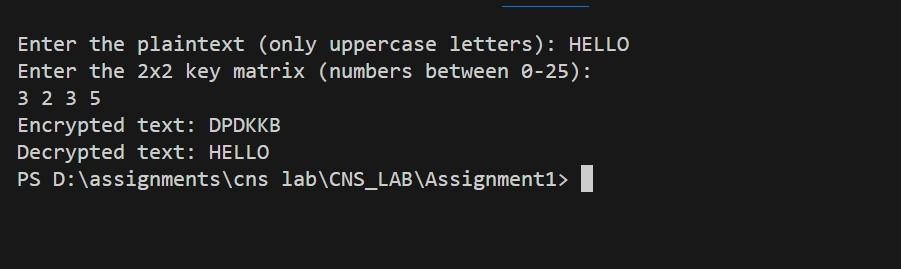
string ciphertext = encrypt(plaintext, keyMatrix, matrixSize); cout << "Encrypted text: " << ciphertext << endl;

// Decrypt the ciphertext

string decryptedText = decrypt(ciphertext, invKeyMatrix, matrixSize); cout << "Decrypted text: " << decryptedText << endl;

return 0;

}

Output:

Advantages:

* Polyalphabetic: Uses linear algebra for encryption, making it stronger than monoalphabetic ciphers.
* Higher Complexity: More resistant to frequency analysis due to the use of matrices.

Disadvantages:

* Complexity: Requires matrix inversion and modular arithmetic, which can be cumbersome.
* Key Size: Key matrix must be invertible, and the length of the plaintext must be a multiple of the matrix size.
  1. Vigenere cipher Ans:

The Vigenère Cipher is a method of encrypting text using a keyword. It works by shifting each letter in the plaintext by an amount determined by the corresponding letter in the keyword. The key repeats itself if it's shorter than the plaintext.

How It Works:

* + 1. Keyword: Choose a keyword (e.g., "KEY").
    2. Encryption:
       - Write the keyword repeatedly above the plaintext. o Shift each letter in the plaintext by the position of the corresponding letter in the keyword (A=0, B=1, ..., Z=25).
    3. Decryption:
       - Use the same keyword to reverse the shifts and recover the plaintext.

Python Code: #include

<iostream> using namespace std;

// Function to encrypt the text using the Vigenère cipher string vigenereEncrypt(string text, string key) { string result = ""; int keyIndex = 0;

for (int i = 0; i < text.length(); i++) {

// Process only alphabetic characters if (isalpha(text[i])) {

char c = toupper(text[i]); // Convert to uppercase

char k = toupper(key[keyIndex % key.length()]); // Key character

char encryptedChar = (c + k - 2 \* 'A') % 26 + 'A'; // Encryption formula

result += encryptedChar;

keyIndex++; // Move to the next character in the key

} else {

result += text[i]; // Non-alphabetic characters remain unchanged

}

}

return result;

}

// Function to decrypt the text using the Vigenère cipher string vigenereDecrypt(string text, string key) { string result = ""; int keyIndex = 0;

for (int i = 0; i < text.length(); i++) {

// Process only alphabetic characters if (isalpha(text[i])) {

char c = toupper(text[i]); // Convert to uppercase

char k = toupper(key[keyIndex % key.length()]); // Key character char decryptedChar = (c - k + 26) % 26 + 'A'; // Decryption formula result += decryptedChar;

keyIndex++; // Move to the next character in the key

} else {

result += text[i]; // Non-alphabetic characters remain unchanged

}

}

return result;

}

int main() { string text; string key;

cout << "Enter text: ";

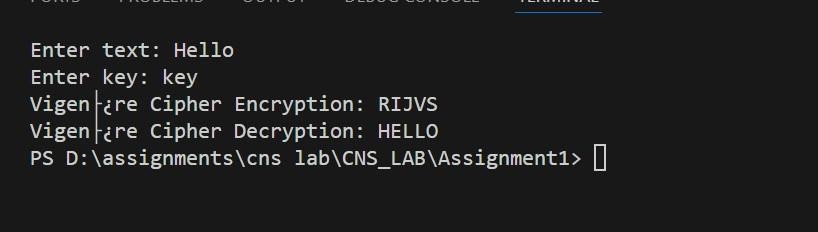
getline(cin, text); // Use getline to allow spaces in input cout << "Enter key: ";

getline(cin, key); // Use getline to allow spaces in the key

string encrypted = vigenereEncrypt(text, key); string decrypted = vigenereDecrypt(encrypted, key);

cout << "Vigenère Cipher Encryption: " << encrypted << endl;

return 0;



* Polyalphabetic: Uses a keyword to shift letters, making it more secure than Caesar Cipher.
* Improved Security: Harder to crack with frequency analysis if the keyword is long and complex.

Disadvantages:

* Keyword Management: Security depends on the keyword length and complexity.
* Vulnerabilities: Can be broken with techniques like the Kasiski examination or frequency analysis if the keyword is short.

cout << "Vigenère Cipher Decryption: " << decrypted << endl; } Output:

Advantages:

#### ASSIGNMENT 2

**:**

#### NAME :

1. Perform encryption and decryption using following transposition techniques
   1. Rail fence Ans:

The Rail Fence Cipher is a type of transposition cipher where the plain text is written in a zigzag pattern across multiple "rails" (rows) and then read row by row to create the cipher text. Decryption involves reconstructing the zigzag pattern to retrieve the original message.

Code c++:

#include <iostream> #include <vector> using namespace std;

string railFenceEncrypt(string text, int key) { vector<string> rail(key); int row = 0; bool down = true;

for (char c : text) { rail[row] += c;

if (row == key - 1) down = false; else if (row == 0) down = true; down

? row++ : row--;

}

string result;

for (string r : rail) result += r; return result;

}

string railFenceDecrypt(string text, int key) { vector<int> rail(key); int row = 0; bool down = true;

for (int i = 0; i < text.length(); i++) { rail[row]++; if (row == key - 1) down = false; else if (row == 0) down = true; down ? row++ :

row--;

}

vector<string> railText(key); int index = 0; for (int i = 0; i < key; i++) { railText[i] = text.substr(index, rail[i]); index += rail[i];

}

string result; row = 0; down

= true;

for (int i = 0; i < text.length(); i++) { result += railText[row][0]; railText[row] = railText[row].substr(1);

if (row == key - 1) down = false; else if (row == 0) down = true; down ? row++ : row--;

}

return result;

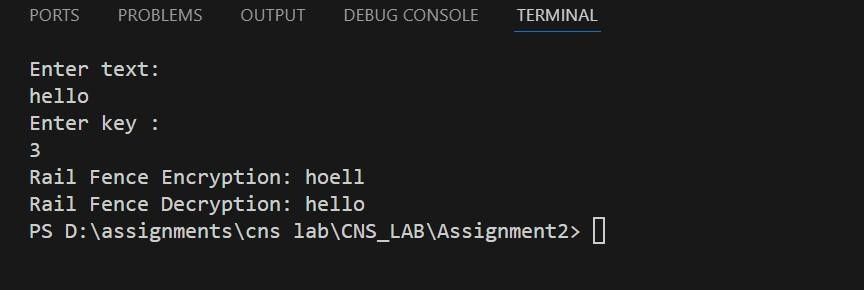
}

int main() { string text; int key;

cout<<"Enter text: "<<endl; cin>>text; cout<<"Enter key : "<<endl; cin>>key;

string encrypted = railFenceEncrypt(text, key); string

return 0;



* Simplicity: Easy to understand and implement.
* Low Computation: Requires minimal computational resources for encryption and decryption.

Disadvantages:

* Weak Security: Very easy to break with simple analysis or known- plaintext attacks.
* Pattern Recognition: The regular zigzag pattern makes it susceptible to pattern recognition, which can be exploited to decode the message.
  1. row and Column Transformation

decrypted = railFenceDecrypt(encrypted, key); cout << "Rail Fence Encryption: " << encrypted << endl; cout << "Rail Fence Decryption: " << decrypted << endl; }

Output:

Advantages:

Ans:

Row and column transformation is a type of transposition cipher where the message is written in a grid (matrix) and the order of rows and columns is changed according to a key.

Row Transposition: Encrypts text by writing it into rows of a grid, then permuting the columns according to a specific key.

Column Transposition: Encrypts text by writing it into columns of a grid, then permuting the rows according to a specific key.

How It Works:

* + 1. Write the plaintext into a grid according to the number of rows or columns.
    2. Permute the rows or columns based on the key.
    3. Read off the text in the new order to get the ciphertext.

c++ code:

#include <iostream> #include <string> #include <vector>

#include <algorithm> // for sort

using namespace std;

// Function to pad the plaintext with 'X' to fill the grid string padText(string plaintext, int cols) { int len = plaintext.length();

int padLen = cols - (len % cols); // Calculate how many characters are needed to fill the last row if (padLen != cols) {

for (int i = 0; i < padLen; i++) { plaintext += 'X'; // Padding with 'X'

}

}

return plaintext;

}

// Function to encrypt the plaintext using Row-Column Transposition Cipher with custom key

string encryptRowColumn(string plaintext, int cols, const vector<int>& key) {

// Padding the plaintext if necessary plaintext = padText(plaintext, cols);

int rows = plaintext.length() / cols; string ciphertext;

// Read columns based on the custom key to form the ciphertext for (int col : key) {

for (int row = 0; row < rows; row++) {

ciphertext += plaintext[row \* cols + (col - 1)]; // col-1 for 0-based indexing

}

}

return ciphertext;

}

// Function to decrypt the ciphertext using Row-Column Transposition Cipher with custom key

string decryptRowColumn(string ciphertext, int cols, const vector<int>& key)

{

int rows = ciphertext.length() / cols;

string plaintext(ciphertext.length(), ' '); // Placeholder for the decrypted text

// Place the ciphertext back in the grid using the custom key int index = 0; for (int col : key) {

for (int row = 0; row < rows; row++) {

plaintext[row \* cols + (col - 1)] = ciphertext[index++]; // col-1 for 0-based indexing

}

}

// Remove any trailing 'X' added during padding while (plaintext.back() == 'X') { plaintext.pop\_back();

}

return plaintext;

}

int main() { string plaintext; int cols; vector<int> key;

// Input the plaintext cout

<< "Enter the plaintext: "; getline(cin, plaintext);

// Input the number of columns cout

<< "Enter the number of columns: "; cin

>> cols;

// Input the custom key for column order

cout << "Enter the key for column order (enter " << cols << " spaceseparated numbers): "; for (int i = 0; i < cols; i++) { int k; cin >> k; key.push\_back(k);

}

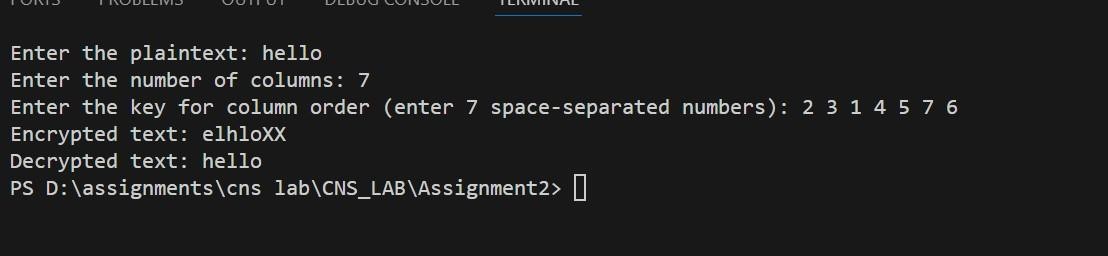
// Encrypt the plaintext

string encryptedText = encryptRowColumn(plaintext, cols, key); cout << "Encrypted text: " << encryptedText << endl;

return 0;

// Decrypt the ciphertext

string decryptedText = decryptRowColumn(encryptedText, cols, key); cout << "Decrypted text: " << decryptedText << endl;



* Increased Security: More complex than simple transpositions.
* Flexibility: Key-based rearrangement can add security. Disadvantages:
* Complexity: Can be more complex to implement and manage compared to simple ciphers.
* Pattern Recognition: Still susceptible to pattern analysis if not combined with other encryption methods.

}

Output:

Advantages:

**:**

**Name:**

# Assignment 3

1. Implementation of Euclidean and Extended Euclidean Algorithm Ans:

The Euclidean and Extended Euclidean algorithms are essential for finding the greatest common divisor (GCD) of two integers. The Extended Euclidean algorithm also finds the coefficients of Bézout's identity, which are useful in solving linear Diophantine equations and in modular arithmetic.

Euclidean Algorithm

The Euclidean algorithm finds the GCD of two numbers by repeatedly applying the following rule: gcd(a, b) = gcd(b, a % b) until b becomes zero. The GCD is then the non-zero remainder.

Extended Euclidean Algorithm

The Extended Euclidean algorithm not only computes the GCD of two integers a and b, but also finds integers x and y such that ax + by = gcd(a, b).

Code C++:

/\*Euclidean and Extended Euclidean Algorithm\*/ #include <iostream>

using namespace std;

int gcd(int a, int b) {

if (b == 0) return a; return gcd(b, a % b);

}

int extendedGCD(int a, int b, int &x, int &y) { if (a == 0) { x = 0; y = 1;

return b;

}

int x1, y1;

int gcd = extendedGCD(b % a, a, x1, y1); x = y1 - (b / a) \* x1;

y = x1; return gcd;

cout<<"Enter greater number :"<<endl; cout<<"Enter lower number :"<<endl;

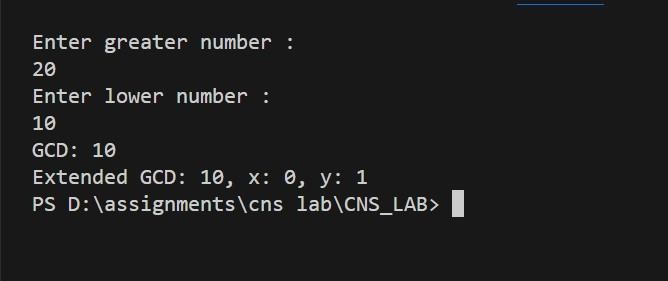
int gcd\_value = gcd(a, b);

int extended\_gcd\_value = extendedGCD(a, b, x, y);

cout << "GCD: " << gcd\_value << endl;

cout << "Extended GCD: " << extended\_gcd\_value << ", x: " << x << ", y: " << y << endl;

return 0;



}

int main() { int a , b ; cin>>a;

cin>>b; int x, y;

}

Output:

This implementation of the Euclidean and Extended Euclidean algorithms is fundamental in cryptography, number theory, and algorithms related to modular arithmetic.

# Assignment 4

1. Implementation of Chinese Remainder Theorem (CRT) Ans:

The Chinese Remainder Theorem (CRT) is a powerful tool in number theory that provides a solution to a system of simultaneous congruences with pairwise coprime moduli. Given a system of congruences, the CRT allows us to find a unique solution modulo the product of the moduli.

Problem Description

Given n congruences: x≡a1 (mod m1), x≡a2 (mod m2) ⋮x ≡ an (mod mn)

Where the moduli m1, m2, …, mn are pairwise coprime, the CRT provides a unique solution modulo M=m1×m2×⋯× mn.

For each congruence x ≡ ai (mod mi), it calculates the partial solution using the formula: x ≡ ai × Mi × inverse(Mi, mi) (mod M) where Mi=M/mi

The final solution is obtained by summing all partial solutions modulo M.

Code C++:

#include <iostream> #include <vector> using namespace std;

// Extended Euclidean Algorithm to find the GCD and the coefficients x and y int extendedGCD(int a, int b, int &x, int &y) {

if (a == 0) {

x = 0;

y = 1; return b;

}

int x1, y1;

int gcd = extendedGCD(b % a, a, x1, y1); x = y1 - (b / a) \* x1; y = x1; return gcd;

}

// Function to find the modular inverse using Extended Euclidean Algorithm int modInverse(int a, int m) { int x, y;

int g = extendedGCD(a, m, x, y); if (g != 1) return -1; // Inverse doesn't exist return (x % m + m) % m;

}

// Function implementing Chinese Remainder Theorem int chineseRemainderTheorem(vector<int> num, vector<int> rem) { int prod = 1; // Product of all numbers for (int i : num)

prod \*= i;

int result = 0; // Result of the system of congruences for (int i = 0; i < num.size(); i++) { int pp = prod / num[i]; // Partial product int inv = modInverse(pp, num[i]); // Modular inverse result

+= rem[i] \* inv \* pp;

}

return result % prod; // Return the result modulo the product

}

int main() { int n;

cout << "Enter the number of equations: "; cin >> n;

vector<int> num(n), rem(n);

// Taking input for modulus values and remainders cout << "Enter the modulus values:\n"; for (int i = 0; i < n; i++) { cout << "Modulus " << i + 1 << ": "; cin >> num[i];

}

cout << "Enter the remainders:\n";

for (int i = 0; i < n; i++) {

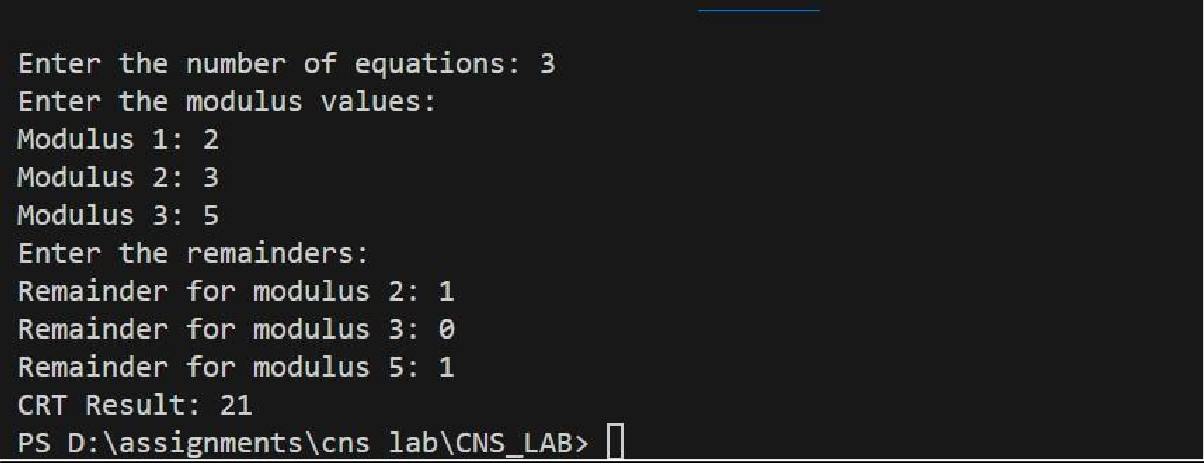
cout << "Remainder for modulus " << num[i] << ": "; cin >> rem[i];

}

// Calculate and display the result of the Chinese Remainder Theorem int result = chineseRemainderTheorem(num, rem); cout << "CRT Result: " << result << endl;

return 0;

}

Output:

# Assignment 5

1. Apply DES algorithm for practical applications Ans:

The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of digital data. Although DES is now considered insecure for many applications due to its small key size, it is still an important algorithm for understanding the basics of cryptography.

Practical Application of DES Algorithm

To apply the DES algorithm in a practical application, we can use the pycryptodome library in Python, which provides an implementation of DES. Below is an example that demonstrates how to use DES to encrypt and decrypt a message.

Python Code: from Crypto.Cipher import DES from Crypto.Util.Padding import pad, unpad from Crypto.Random import get\_random\_bytes

def des\_encrypt(plain\_text, key): """

Encrypt the plain text using DES algorithm. Parameters:

plain\_text (str): The text to be encrypted.

key (bytes): The encryption key (must be 8 bytes long).

Returns:

bytes: The encrypted cipher text. """

cipher = DES.new(key, DES.MODE\_ECB)

padded\_text = pad(plain\_text.encode(), DES.block\_size) encrypted\_text = cipher.encrypt(padded\_text) return encrypted\_text

def des\_decrypt(cipher\_text, key): """

Decrypt the cipher text using DES algorithm.

Parameters:

cipher\_text (bytes): The encrypted text to be decrypted. key (bytes): The decryption key (must be 8 bytes long).

Returns:

str: The decrypted plain text. """

cipher = DES.new(key, DES.MODE\_ECB)

decrypted\_text = unpad(cipher.decrypt(cipher\_text), DES.block\_size) return decrypted\_text.decode()

def main(): """

The main function to run the program. """

print("\nDES Encryption and Decryption")

# Generate a random 8-byte key for DES key = get\_random\_bytes(8)

print(f"\nGenerated Key (in hexadecimal): {key.hex()}")

# Input plaintext

plain\_text = input("Enter the plain text to encrypt: ")

# Encrypt the plaintext

encrypted\_text = des\_encrypt(plain\_text, key)

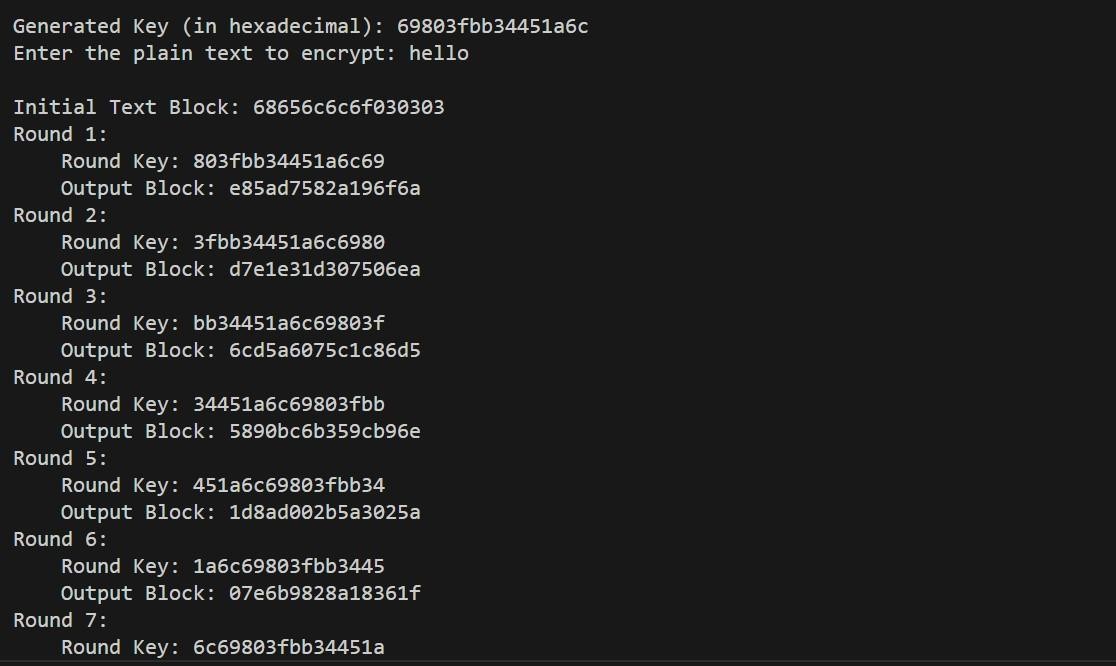
print(f"\nEncrypted Text (in hexadecimal): {encrypted\_text.hex()}")

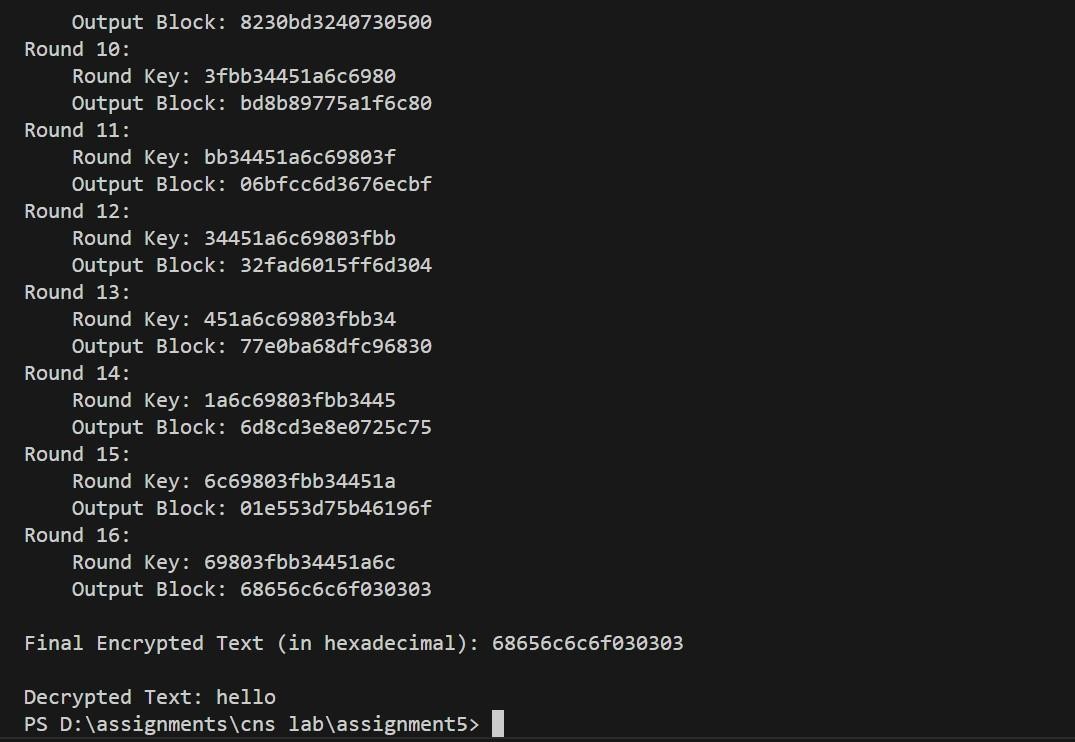
# Decrypt the ciphertext

decrypted\_text = des\_decrypt(encrypted\_text, key) print(f"\nDecrypted Text: {decrypted\_text}")

if name == " main ": main()

Output:





Practical Applications:

* File Encryption: DES can be used to encrypt sensitive files before storing them in insecure locations.
* Secure Communication: DES ensures that messages sent over a network are unreadable to unauthorized parties.
* Password Storage: Encrypting passwords before storing them in databases (though modern standards recommend stronger algorithms like AES).

While DES itself is outdated and not recommended for secure applications, understanding how it works is crucial for grasping more advanced encryption algorithms like AES.

# Assignment 6

### :

1. Apply AES algorithm for practical applications Ans:

The Advanced Encryption Standard (AES) is a widely used symmetric encryption algorithm that is both fast and secure. It is the standard encryption algorithm used by governments, financial institutions, and many other organizations. Unlike DES, which is now considered insecure, AES is robust and provides a high level of security.

Practical Application of AES Algorithm

We can use the pycryptodome library in Python to implement AES encryption and decryption. The AES algorithm can work with key sizes of 128, 192, or 256 bits, and it operates on 128-bit blocks. In this example, we'll use AES with a 256-bit key in Cipher Block Chaining (CBC) mode.

Python Code:

from Crypto.Cipher import AES from Crypto.Util.Padding import pad, unpad from Crypto.Random import get\_random\_bytes

def aes\_encrypt(plain\_text, key): """

Encrypt the plain text using AES algorithm.

Parameters:

plain\_text (str): The text to be encrypted. key (bytes): The encryption key (must be 16, 24, or 32 bytes long).

Returns:

bytes: The initialization vector (IV) and the encrypted cipher text. """

cipher = AES.new(key, AES.MODE\_CBC) iv = cipher.iv # Initialization vector padded\_text = pad(plain\_text.encode(), AES.block\_size)

encrypted\_text = cipher.encrypt(padded\_text) return iv, encrypted\_text

def aes\_decrypt(iv, cipher\_text, key):

"""

Decrypt the cipher text using AES algorithm.

Parameters:

iv (bytes): The initialization vector used during encryption. cipher\_text (bytes): The encrypted text to be decrypted. key (bytes): The decryption key (must be 16, 24, or 32 bytes long).

Returns: str: The decrypted plain text.

"""

cipher = AES.new(key, AES.MODE\_CBC, iv) decrypted\_text = unpad(cipher.decrypt(cipher\_text), AES.block\_size) return decrypted\_text.decode()

def main():

"""

The main function to run the program. """

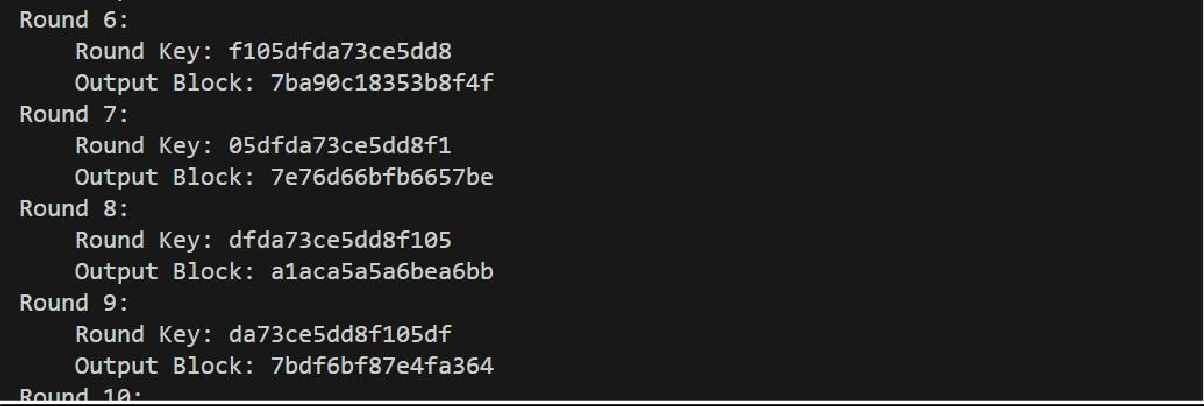
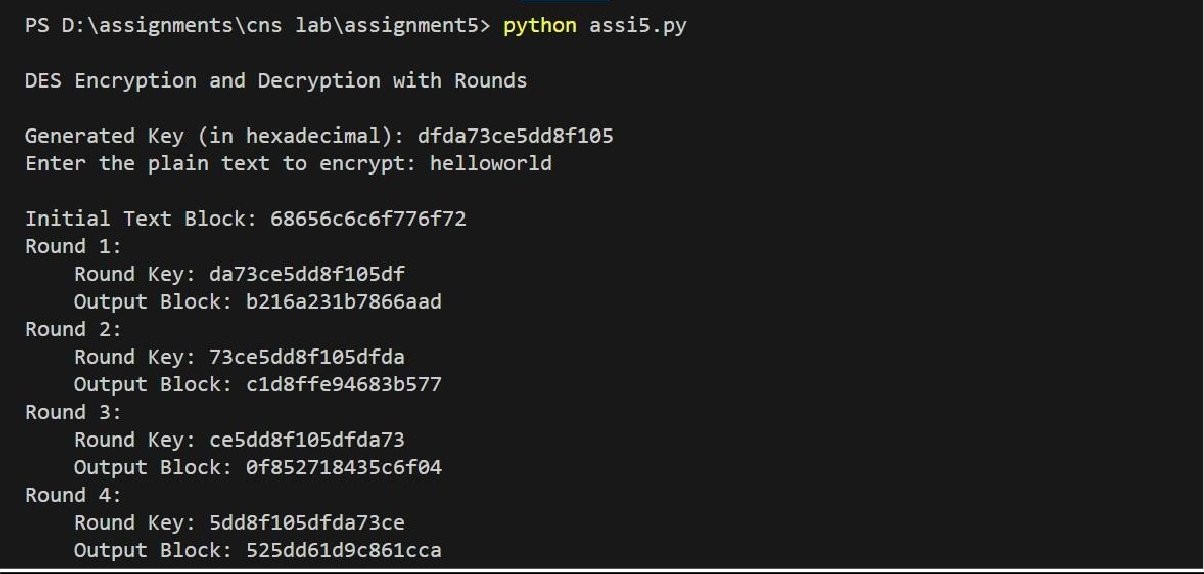
print("\nAES Encryption and Decryption")

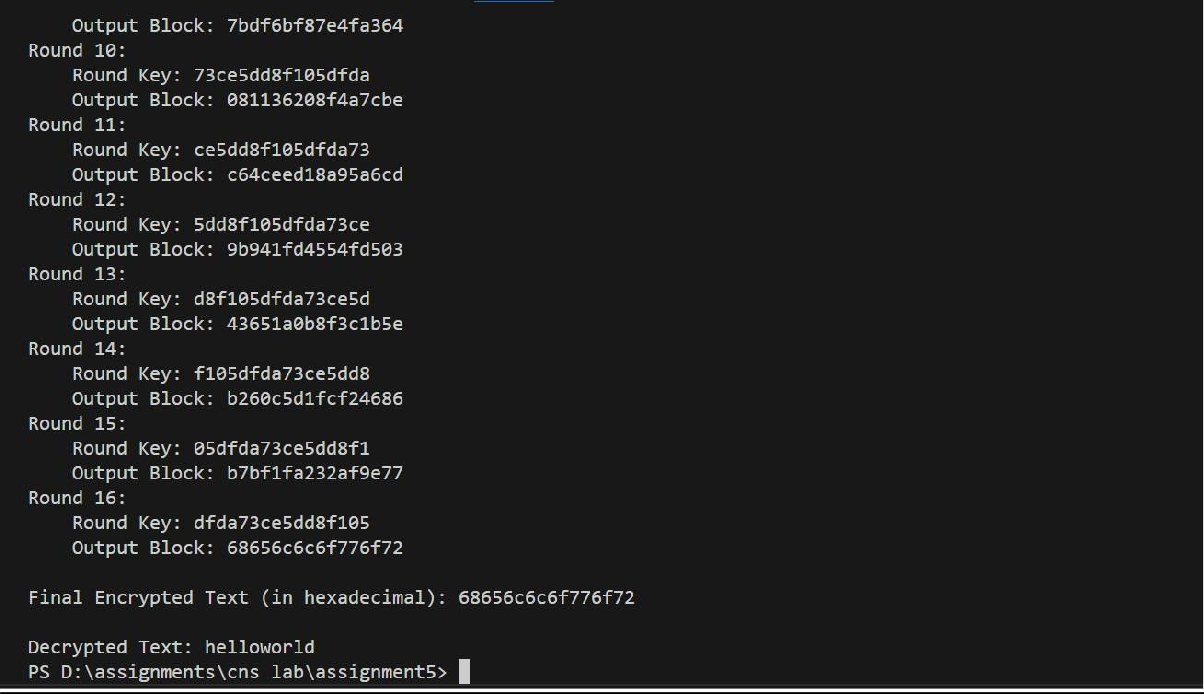
# Generate a random 32-byte key for AES (256-bit) key = get\_random\_bytes(32) print(f"\nGenerated Key (in hexadecimal): {key.hex()}")

# Input plaintext plain\_text = input("\nEnter the plain text to encrypt: ")

# Encrypt the plaintext iv, encrypted\_text = aes\_encrypt(plain\_text, key) print(f"\nInitialization Vector (IV) (in hexadecimal): {iv.hex()}") print(f"\nEncrypted Text (in hexadecimal): {encrypted\_text.hex()}") # Decrypt the ciphertext decrypted\_text = aes\_decrypt(iv, encrypted\_text, key) print(f"\nDecrypted Text: {decrypted\_text}")

if name == " main ": main()

Output:



Practical Applications of AES:

* File Encryption: Encrypting sensitive files before storing them on disk.
* Secure Communication: Ensuring that data sent over the network remains confidential.
* Data Protection in Applications: Encrypting user data, such as passwords, to protect them from unauthorized access.

AES is widely adopted due to its strength and efficiency, and it remains the standard for securing digital data across various industries.

CNS LA-2

Assignment -7 Implement RSA Algorithm

#include <bits/stdc++.h> #define ll int128\_t #define ld long double

#define fast ios\_base::sync\_with\_stdio(false);cin.tie(NULL);cout.tie(NULL); using namespace std;

ll mulm(ll a, ll b, ll mod) { ll res = 0; a = a % mod;

while (b > 0) { if (b & 1) res = (res + a) % mod; a

= (a \* 2) % mod; b >>= 1;

}

return res % mod;

}

ll powr(ll a, ll b, ll mod) { ll res = 1; while (b) { if (b & 1) res =

mulm(res, a, mod); a = mulm(a, a, mod);

b /= 2;

}

return res;

}

ll inv(ll e, ll phi) { ll t = 0, newt = 1; ll r = phi, newr

= e; while (newr != 0) { ll quotient = r / newr;

t = t - quotient \* newt; swap(t, newt); r = r - quotient \* newr;

swap(r, newr);

}

if (t < 0) t += phi; return t;

}

void print128(ll n) { if (n == 0) {

cout << "0"; return;

}

string res = ""; while (n > 0) { res = char('0' + n % 10) + res;

n /= 10;

}

cout << res;

}

int main() { fast;

ll p = 1000000007, q = 1000000009;

ll n = p \* q; ll phi = (p - 1) \* (q - 1);

ll e = 65537; ll d = inv(e, phi);

cout << "Public Key (n, e): (";

print128(n); cout << ", " << (long long)e

<< ")" << endl;

cout << "Private Key (d): "; print128(d); cout << endl;

ll message = 86897874; cout << "Original Message: " << (long long)message << endl;

ll ciphertext = powr(message, e, n); cout << "Encrypted Ciphertext: ";

print128(ciphertext); cout << endl;

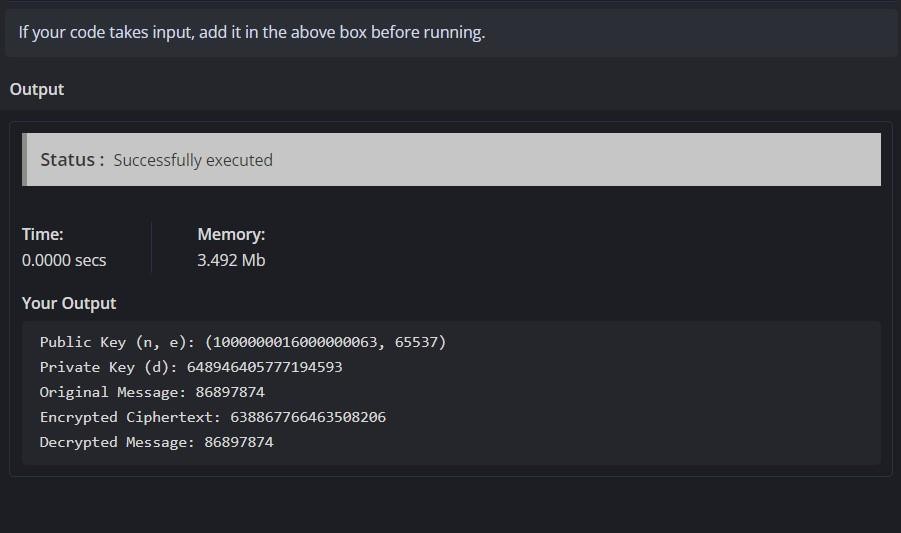
ll decrypted\_message = powr(ciphertext, d, n); cout << "Decrypted Message: ";

print128(decrypted\_message); cout << endl;

return 0;

}

Output:



How RSA Works ?

RSA (Rivest–Shamir–Adleman) is one of the most widely used public-key cryptosystems. It is primarily used for secure data transmission and digital signatures. RSA relies on the mathematical difficulty of factoring large composite numbers and involves key generation, encryption, and decryption steps.

Here’s how RSA works, broken down into key concepts:

1. **Key Generation**

RSA involves two keys:

* + **Public Key (used for encryption)**: Shared with everyone.
  + **Private Key (used for decryption)**: Kept secret by the owner.

**Steps for Key Generation:**

1. **Select two large prime numbers, p and q**:
   * These numbers should be random, and large enough (e.g., 2048-bit) to make the system secure. 2. **Compute n (the modulus)**:

n=p×qn = p \times qn=p×q

* + n is used as part of both the public and private keys.
  + n is also the modulus for both encryption and decryption operations.

1. **Compute Euler's Totient function, φ(n)**:

φ(n)=(p−1)×(q−1)φ(n) = (p - 1) \times (q - 1)φ(n)=(p−1)×(q−1)

* + This value is used in the process of creating the private key.

1. **Choose an encryption exponent e**:
   * e is the public key exponent and should be relatively prime to φ(n) (i.e., gcd(e, φ(n))

= 1).

* + A commonly chosen value is e = 65537, as it balances efficiency and security.

1. **Calculate the decryption exponent d**:
   * d is the private key exponent, and it's calculated as the modular inverse of e mod φ(n):

d×e≡1 (mod φ(n))d \times e \equiv 1 \ (\text{mod}\ φ(n))d×e≡1 (mod φ(n))

* + In other words, d is the number such that when multiplied by e, the result is congruent to 1 modulo φ(n).

1. **Public and Private Keys**:
   * **Public Key**: Consists of (e, n).
   * **Private Key**: Consists of (d, n).
2. **Encryption (Using the Public Key)**

To encrypt a message M (as a number) using the recipient’s public key (e, n):

1. Convert the plaintext message M into a number.
2. Compute the ciphertext C using the following formula: C=Me (mod n)C = M^e \ (\text{mod}\ n)C=Me (mod n)

o This operation makes it computationally infeasible to retrieve the original message without the private key, due to the difficulty of factoring n.

1. **Decryption (Using the Private Key)**

To decrypt the ciphertext C using the recipient’s private key (d, n):

* 1. Compute the original message M using the formula: M=Cd (mod n)M = C^d \ (\text{mod}\ n)M=Cd (mod n)

o This restores the original message, as the relationship between e, d, and n ensures that decryption is the inverse of encryption.

## Assignment 8: Implement Diffi-Hellman Key Exchange Algorithm

#include <iostream> #include <iomanip> using namespace std;

#define ll int128

ll mulm(ll a, ll b, ll mod) { ll res = 0;

a = a % mod; while (b > 0) { if (b & 1) res

= (res + a) % mod; a = (a \* 2) % mod; b >>= 1;

}

return res % mod;

}

ll powr(ll a, ll b, ll mod) { ll res = 1; while (b) { if (b & 1) res =

mulm(res, a, mod); a = mulm(a, a, mod);

b /= 2;

}

return res;

}

void print128(ll n) { if (n == 0) {

cout << "0"; return;

}

string res = ""; while (n > 0) { res = char('0' + n % 10) + res;

n /= 10;

}

cout << res;

}

int main() {

ios\_base::sync\_with\_stdio(false); cin.tie(NULL);

ll p = 100000000000000000003;

ll g = 5;

ll a = 987654321098765432; ll b = 876543210987654321;

ll A = powr(g, a, p);

ll B = powr(g, b, p);

ll S\_A = powr(B, a, p); ll S\_B = powr(A, b, p);

cout << "Publicly shared values: " << endl; cout << "Prime p: "; print128(p); cout << endl; cout << "Primitive root g: ";

print128(g); cout << endl;

cout << "\nPrivate keys:" << endl; cout << "Alice's private key: ";

print128(a); cout << endl; cout

<< "Bob's private key: ";

print128(b); cout << endl;

cout << "\nPublic keys exchanged:" << endl; cout << "Alice's public key (A): ";

print128(A); cout << endl; cout << "Bob's public key (B): "; print128(B); cout <<

endl;

cout << "\nShared secret calculated by Alice (S\_A): "; print128(S\_A); cout << endl; cout << "Shared secret calculated by Bob (S\_B): "; print128(S\_B); cout << endl;

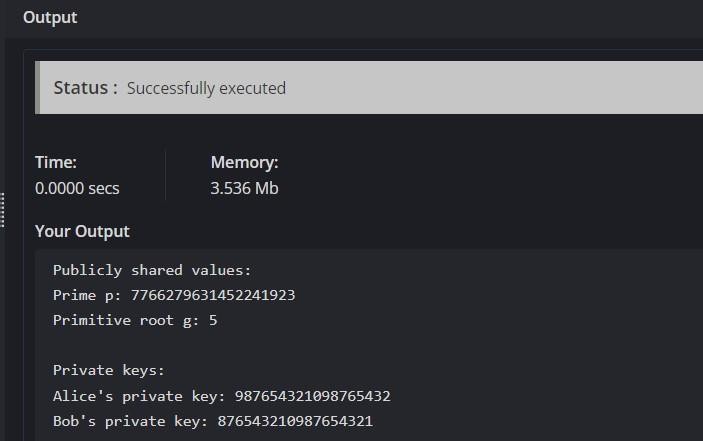
if (S\_A == S\_B) { cout << "\nKey exchange successful. Shared secret: "; print128(S\_A); cout << endl; } else { cout << "\nKey exchange failed." << endl;

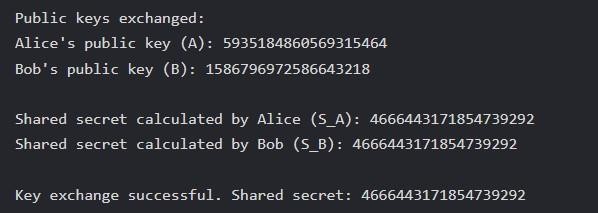
}

return 0;

}

Output:





## Assignment-9: Calculate the message digest of a text using the SHA-1 algorithm

import hashlib

def calculate\_sha1\_digest(text):

# Create a SHA-1 hash object sha1\_hash = hashlib.sha1()

# Update the hash object with the bytes of the text sha1\_hash.update(text.encode('utf- 8'))

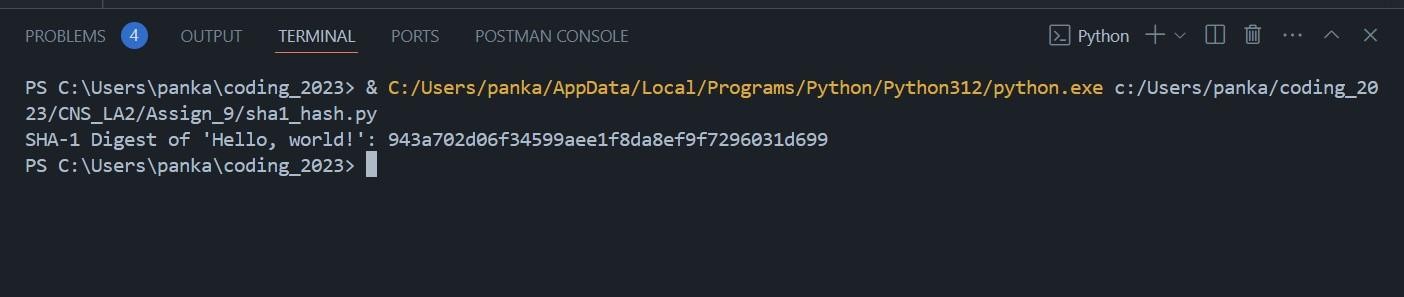
# Get the hexadecimal representation of the digest digest = sha1\_hash.hexdigest()

return digest

# Example usage

text = "Hello, world!" digest =

calculate\_sha1\_digest(text) print(f"SHA- 1 Digest of '{text}': {digest}")

Output:

## Assignment 10: Digital Signature Standard(DSS)

from cryptography.hazmat.backends import default\_backend from cryptography.hazmat.primitives.asymmetric import dsa from cryptography.hazmat.primitives.asymmetric import utils from

cryptography.hazmat.primitives import hashes

# Function to generate a key pair def generate\_key\_pair():

private\_key = dsa.generate\_private\_key(key\_size=2048, backend=default\_backend()) public\_key = private\_key.public\_key() return private\_key, public\_key

# Function to sign a message def sign\_message(private\_key, message): # Hash the message using SHA-256 digest =

hashes.Hash(hashes.SHA256(), backend=default\_backend()) digest.update(message.encode('utf-8')) message\_digest = digest.finalize()

# Generate the signature using the private key and specify the hash algorithm signature = private\_key.sign( message\_digest,

utils.Prehashed(hashes.SHA256()) # Specify the pre-hashed algorithm

)

return signature

# Function to verify a signature def verify\_signature(public\_key, message, signature): # Hash the message again using SHA-256

digest = hashes.Hash(hashes.SHA256(), backend=default\_backend()) digest.update(message.encode('utf-8'))

message\_digest = digest.finalize()

# Verify the signature try:

public\_key.verify( signature,

message\_digest,

utils.Prehashed(hashes.SHA256()) # Use the same pre-hashed algorithm

)

return True # Signature is valid except Exception:

return False # Signature is invalid

# Example usage if name == " main ": # Generate key pair private\_key, public\_key = generate\_key\_pair()

# Define a message message = "This is a test message."

# Sign the message signature = sign\_message(private\_key, message) print(f"Signature: {signature.hex()}")

# Verify the signature is\_valid =

verify\_signature(public\_key, message, signature) print(f"Is the signature valid? {is\_valid}")

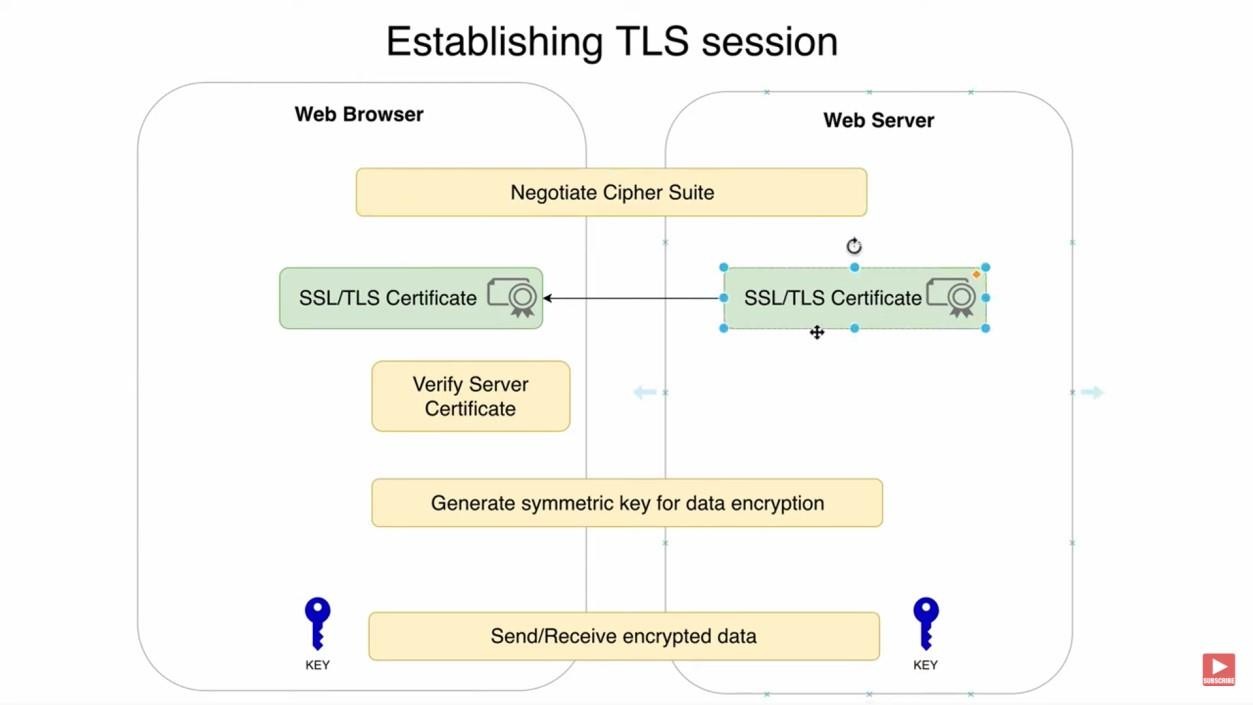
Output:



## Assignment 11: Demonstration of SSL/TLS Using Wireshark

For demonstration I accessed : <https://www.youtube.com/watch?v=MQg48n9lV0s>

By [Bogdan Stashchuk](https://www.youtube.com/%40Bogdan_Stashchuk)



**Transport Layer Security (TLS)**

**Transport Layer Security (TLS)** is a cryptographic protocol designed to provide secure

communication over a computer network. It is widely used in applications such as web browsing,

email, and VoIP to ensure privacy, data integrity, and secure connections between clients and servers.

TLS is the successor to **Secure Sockets Layer (SSL)**, and its primary goal is to prevent eavesdropping, tampering, and message forgery during data transmission.

**Key Features of TLS**

1. **Confidentiality**: TLS ensures that any data sent between two parties (client and server) is encrypted, making it unreadable to third parties.
2. **Data Integrity**: TLS ensures that data cannot be modified or corrupted without being detected during transmission.
3. **Authentication**: TLS allows both parties (typically the client and server) to authenticate each other. Servers typically authenticate using digital certificates issued by trusted Certificate Authorities (CAs).

**How TLS Works**

TLS operates in two main phases:

1. **TLS Handshake**: This is the initial setup phase where the client and server agree on encryption algorithms, exchange keys, and authenticate each other.
2. **Secure Data Transfer**: After the handshake, all data is encrypted using symmetric encryption for efficient and secure communication.

**1. TLS Handshake**

During the TLS handshake, the client and server negotiate the encryption parameters and establish a shared secret that will be used to encrypt data. The handshake typically involves the following steps:

1. **Client Hello**:
   * The client sends a "Hello" message to the server, which includes:

 The version of TLS it supports.

 A list of cryptographic algorithms (cipher suites) it supports.

 A random value (nonce).

1. **Server Hello**:
   * The server responds with its own "Hello" message, which includes:

 The TLS version and cipher suite it has chosen from the client’s list.

 Its own random value.

 The server's digital certificate (for server authentication).

1. **Server Authentication**:
   * The server’s digital certificate (typically an X.509 certificate) contains the server’s public key and is signed by a trusted Certificate Authority (CA). The client verifies the certificate to authenticate the server.
2. **Key Exchange**:
   * The client generates a **pre-master secret**, which will be used to derive the shared encryption key.
   * The client encrypts the pre-master secret with the server’s public key (from the certificate) and sends it to the server.
   * The server decrypts the pre-master secret using its private key.
   * Both the client and server use the pre-master secret and the previously exchanged random values to generate the same **session keys** (for symmetric encryption).
3. **Client and Server Finished**:
   * Both parties send a message to each other to confirm that the handshake is complete and that secure communication can begin.
4. **Secure Data Transfer**

Once the handshake is complete, the client and server use the session keys (generated during the handshake) to encrypt and decrypt all data sent between them using **symmetric encryption**. This ensures that data is secure and private during the session.

**TLS Cipher Suites**

A **cipher suite** defines the set of algorithms used in a TLS connection, including:

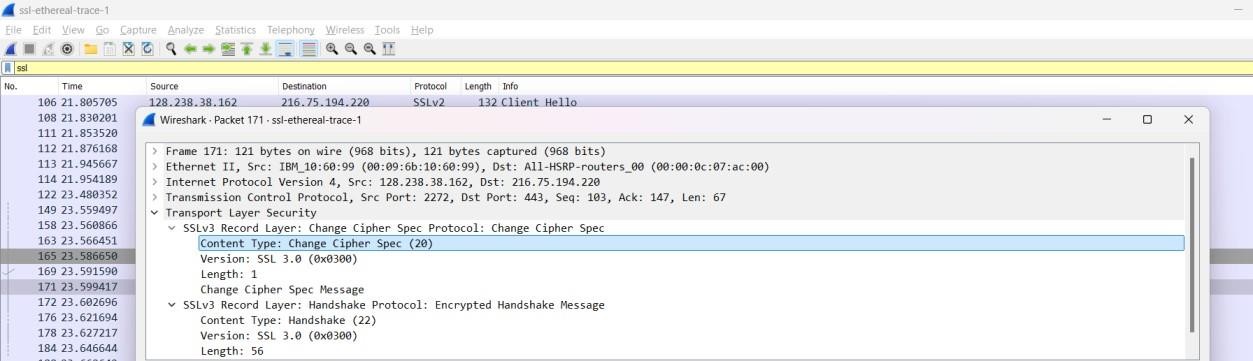
* 1. **Key Exchange Algorithm**: (e.g., RSA, Diffie-Hellman) - Determines how the keys are exchanged.
  2. **Symmetric Encryption Algorithm**: (e.g., AES, ChaCha20) - Encrypts the data during transmission.
  3. **Hashing Algorithm**: (e.g., SHA-256) - Ensures data integrity through the use of message digests.

**TLS Versions**

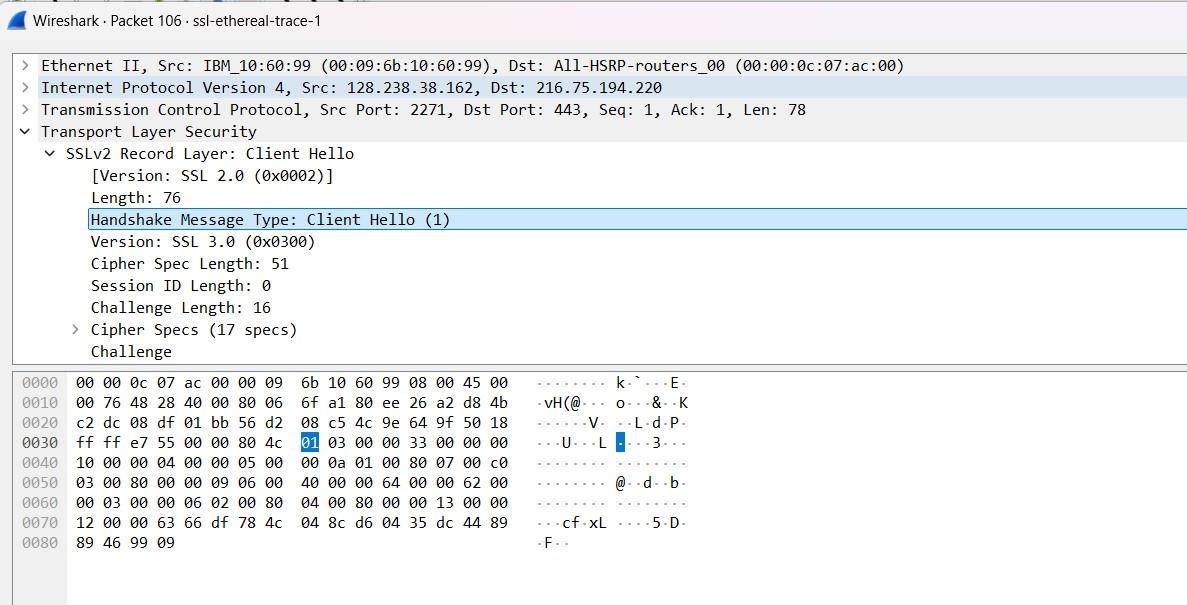
The most widely deployed versions of TLS are:

* **TLS 1.0**: The original version, now deprecated due to security vulnerabilities.
* **TLS 1.1**: Introduced improvements but is also deprecated.
* **TLS 1.2**: Still widely used today, it introduced stronger cryptographic algorithms and more secure hashing.
* **TLS 1.3**: The latest version, offering significant performance improvements and enhanced security, by removing outdated cryptographic techniques and reducing the number of handshake steps. **TLS vs. SSL**
* **SSL (Secure Sockets Layer)** was the original protocol developed by Netscape in the 1990s to secure web communication.
* **TLS (Transport Layer Security)** is the improved version of SSL and is the protocol used today. SSL versions (SSL 2.0 and SSL 3.0) are now deprecated due to security flaws.

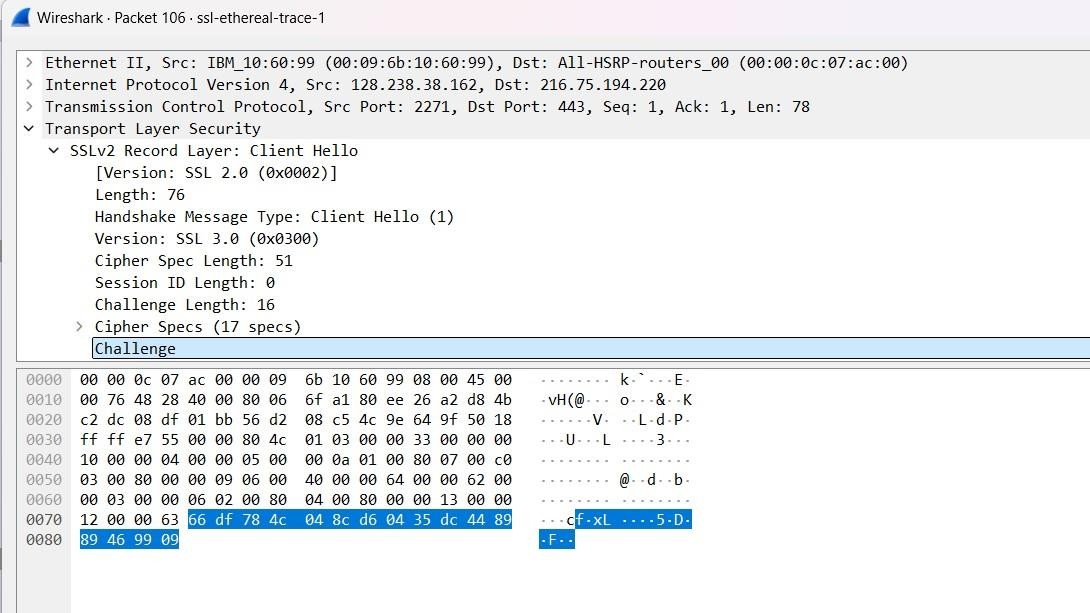
Q2. Each of the SSL records begins with the same three fields (with possibly different values). One of these fields is “content type” and has length of one byte. List all three fields and their lengths.

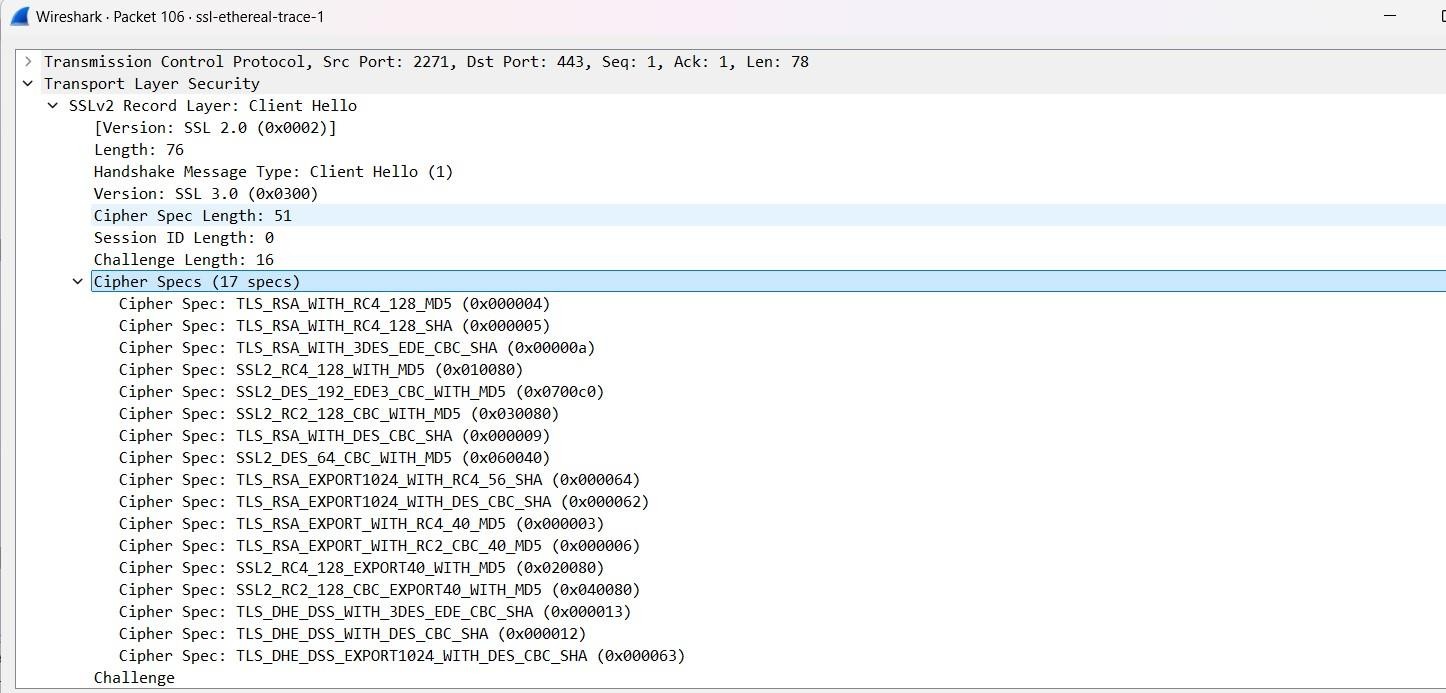


* + Content Type, Version, Length

Q3. Expand the ClientHello record. (If your trace contains multiple ClientHello records, expand the frame that contains the first one.) What is the value of the content type?

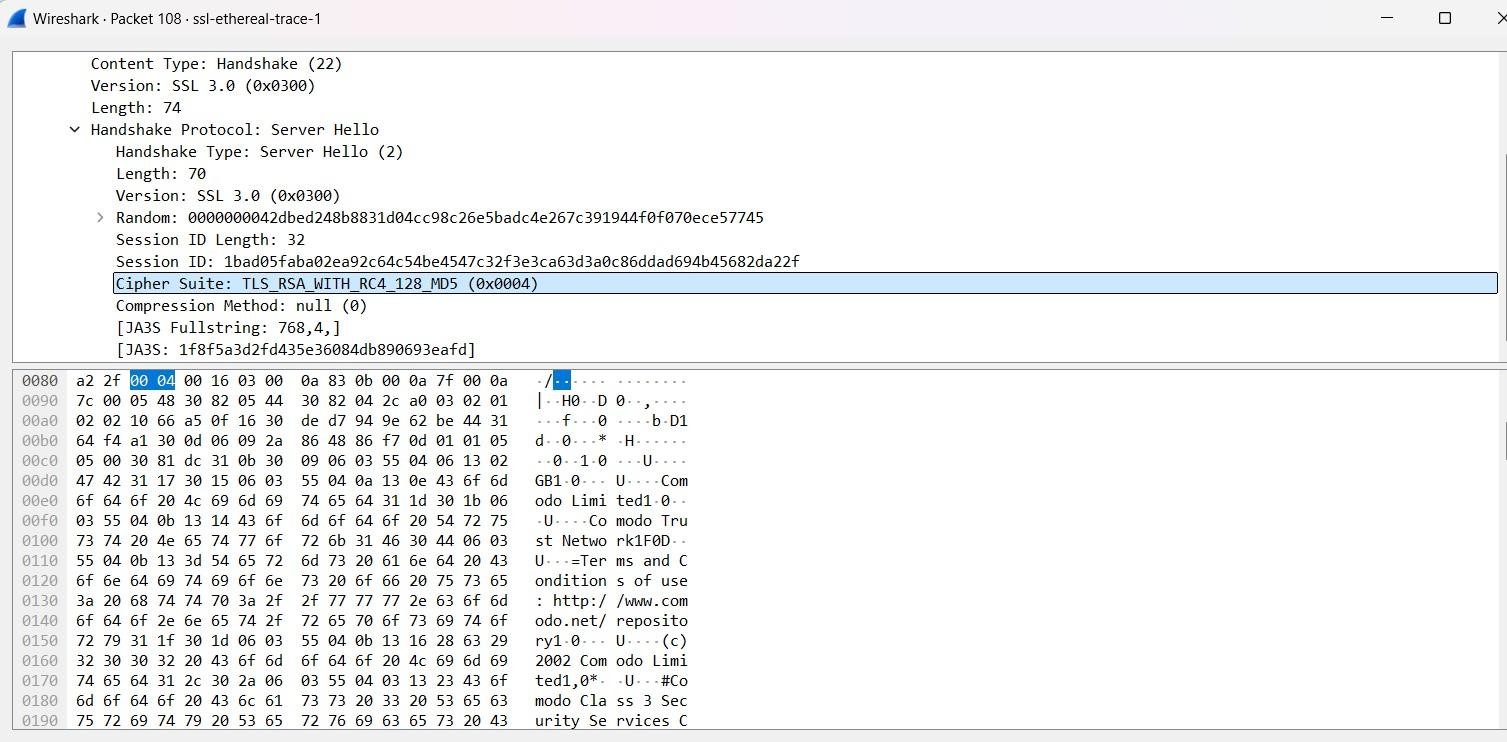
Q4. Does the ClientHello record contain a nonce (also known as a “challenge”)? If so, what is the value of the challenge in hexadecimal notation?



Q5. Does the ClientHello record advertise the cyber suites it supports? If so, in the first listed suite, what are the public-key algorithm, the symmetric-key algorithm, and the hash algorithm?

* + Yes, Client Hello advertises the cyber suites it supports. The first listed suite uses RSA for public-key algorithm with RC-4 128 MD5 as hashing algorithm.

Q6. Locate the ServerHello SSL record. Does this record specify a chosen cipher suite? What are the algorithms in the chosen cipher suite?



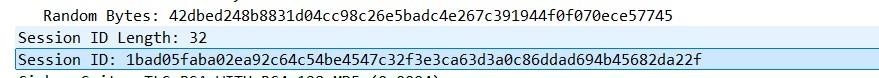
* + The algorithms chosen for cipher suites are RSA algorithm and RC4 128 MD5.

1. Does this record include a nonce? If so, how long is it? What is the purpose of the client and server nonces in SSL?



* + Yes, this record does include a nonce listed under Random. The nonce is 32 bits long, 28 for data and 4 for the time. The purpose is to prevent a replay attack.

Q8. Does this record include a session ID? What is the purpose of the session ID?



* + Yes it does. It provides a unique persistent identifier for the SSL session which is sent in the clear. The client may resume the same session ater by using the server provided session ID when it sends the ClientHello.

Q9. Does this record contain a certificate, or is the certificate included in a separate record. Does the certificate fit into a single Ethernet frame?

* + There is no certificate, it is in another record. It does fit into a single Ethernet frame.

Q10. Locate the client key exchange record. Does this record contain a pre-master secret? What is this secret used for? Is the secret encrypted? If so, how? How long is the encrypted secret?

* + Yes, it does contain a premaster secret. It is used by both the server and client to make a master secret, which is used to generate session keys for MAC and encryption. The secret gets encrypted using the server’s public key, which the client extracted from the certificate sent by the server. The secret is 128 bytes long.

Q11. What is the purpose of the Change Cipher Spec record? How many bytes is the record in your trace?

* + The purpose of the change cipher spec record is to indicate that the contents of the following SSl records sent by the client will be encrypted. This record is 6 bytes long: 5 for the header and 1 for the message segment.

Q12. In the encrypted handshake record, what is being encrypted? How?

* + In the encrypted handshake record, a MAC of the concatenation of all the previous handshake messages sent from this client is generated and sent to the server.

Q13. Does the server also send a change cipher record and an encrypted handshake record to the client? How are those records different from those sent by the client?

* + Yes the server will also send a change cipher spec record and encrypted handshake to the client. The server’s encrypted handshake record is different from that sent by the client because it contains the concatenation of all the handshake messages sent from the server rather than from the client. Otherwise the records would end up being the same.

Q14. How is the application data being encrypted? Do the records containing application data include a MAC? Does Wireshark distinguish between the encrypted application data and the MAC?

* + Application data is encrypted using symmetric key encryption algorithm chosen in the handshake phase (RC4) using the keys generated using the pre-master key and nonces from both client and server. The client encryption key is used to encrypt the data being sent from client to server and the server encryption key is used to encrypt the data being sent from the server to the client.

Q15. Comment on and explain anything else that you found interesting in the trace.

* + The version of SSL used changes from SSLv2 in the initial ClientHello message to SSLv3 in all following message exchanges.
  + Also, during resumes the handshake process is slightly different from the initial one. The client does not need another cert so the server never sends it. It just has to send a new nonce followed by Change Cipher Spec and Encrypted Handshake records from the server to client. After a response from the client then application data can be sent.